(2)

MTL TR 92-32

AD

INSTRUMENT DRIVER FOR GENERAL PURPOSE INTERFACE BUS (IEEE-488)

BRADLEY M. TABER III and PATRICK J. SINCEBAUGH
MATERIALS TESTING AND EVALUATION BRANCH

May 1992

Approved for public release; distribution unlimited.

Original contains color plates: All DTIC reproductions will be in black and white

US ARMY
LABORATORY COMMAND
MATERIALS TECHNOLOGY LABORATORY

U.S. ARMY MATERIALS TECHNOLOGY LABORATORY
Watertown, Massachusetts 02172-0001

DISCLAIMER

The authors and publisher make no warranty of any kind, expressed or implied, with regard to the use of these instrument drivers. In no event will the authors or the U.S. Department of Defense or its affiliates be liable to the user or any third party for direct, indirect, special, incidental, or consequential damages arising out of the use of this software package or the manual.

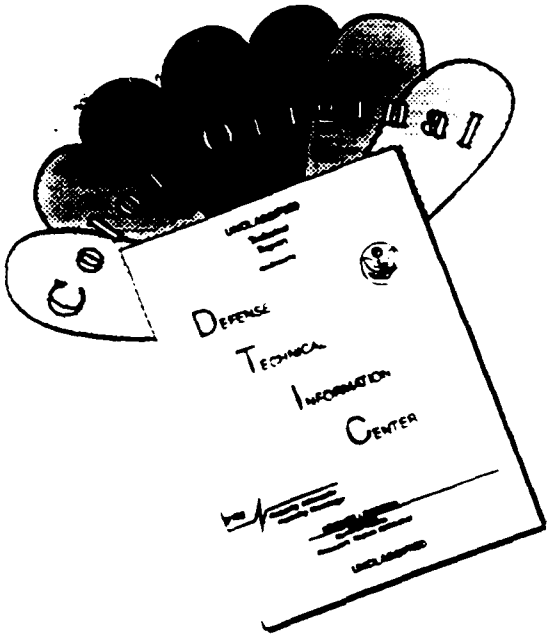
The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

Mention of any trade names or manufacturers in this report shall not be construed as advertising nor as an official indorsement or approval of such products or companies by the United States Government.

DISPOSITION INSTRUCTIONS

Destroy this report when it is no longer needed.
Do not return it to the originator.

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MTL TR 92-32	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) INSTRUMENT DRIVER FOR GENERAL PURPOSE INTERFACE BUS (IEEE-488)		5. TYPE OF REPORT & PERIOD COVERED Final Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bradley M. Taber III and Patrick J. Sincebaugh		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS U.S. Army Materials Technology Laboratory Watertown, Massachusetts 02172-0001 ATTN: SLCMT-MRM		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS D/A Project 1L16105AH84
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Laboratory Command 2800 Powder Mill Road Adelphi, Maryland 20783-1145		12. REPORT DATE May 1992 13. NUMBER OF PAGES 49
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

General purpose interface bus (GPIB)	Instrumentation
Interfaces	Software driver
Data acquisition	Automation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(SEE REVERSE SIDE)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block No. 20

ABSTRACT

Laboratory automation is invaluable to scientists, engineers, and technicians. It provides a mechanism by which many processes can be interactively controlled. A popular method of connecting and controlling laboratory instrumentation is through the General Purpose Interface Bus (GPIB). The Institute of Electrical and Electronics Engineers (IEEE) standardized this bus under IEEE-488. The objective of this project was to study the IEEE-488 and write instrument drivers in C++ for the GPIB and the Tektronix 2430A Digital Oscilloscope. These drivers contain functions for initialization, command control, and data acquisition. The first section of this paper discusses the IEEE-488 and provides pertinent technical information. The second section of this paper is an instruction manual for the use of the GPIB and Tektronix oscilloscope instrument drivers. Portions of an Input/Output (I/O) module are included to provide a graphical representation for the data acquired with the GPIB and Tektronix modules. A test program is presented which demonstrates the use of these modules.

These drivers will be specifically used for ultrasonic data acquisition capabilities for nondestructive testing of materials. The ultrasonic data is generated by an ultrasonic transducer sending a sound pulse through a material. The reflected signal is displayed on a digitizing oscilloscope. The displayed signal must be acquired by the computer for further processing. This system will eventually incorporate robotics for unattended material testing. Human interaction will be kept to a minimum so it will be very important that errors are appropriately handled.

CONTENTS

SECTION 1
IEEE-488 REFERENCE PAPER

INTRODUCTION TO THE IEEE-488

Laboratory researchers have shown great increases in productivity by utilizing automation. As the potential for automated systems became apparent, it was clear that a standard communications interface was necessary. Problems were occurring because the interfacing capabilities would vary between manufacturers. This made most automated systems very customized and non-portable. Hewlett-Packard helped industry overcome this problem by developing the Hewlett-Packard Interface Bus (HPIB) in the early 1970's. This interface bus was widely used throughout industry, and in 1975 became standardized by the Institute of Electrical and Electronic Engineers under Standard IEEE-488. In 1978 the standard was updated with minor revisions. The General Purpose Interface Bus (GPIB), the ANSI MC1.1-1975, and the HPIB are synonymous with the IEEE-488. In this paper, IEEE-488 is used to refer to the standard, and the rules and syntax of communication. GPIB is used to refer to the physical bus or physical devices. The European standards IEC-625.1 and B.S.6146 are very similar to the IEEE-488, the major difference being the type of connector.

The IEEE-488 communications protocol has some distinct advantages, and disadvantages, over more common forms of input/output (I/O) communications such as serial and parallel ports. The major difference is that the GPIB is a bus, rather than a port. The bus protocol allows for the connection of up to fifteen devices, each individually or group addressable. It also allows for a device other than a specific computer to be the active controller, and any device can be set as a talker, listener, or idle. In contrast, there are typically only two or three serial and parallel ports in a PC or workstation. Although there are devices available which will allow a greater number of serial and parallel devices to be attached to one computer, this is usually expensive and inefficient. Because the GPIB is specifically made for instrumentation I/O, it has been a very popular alternative. Most laboratory equipment and instrumentation, and many computer peripherals, such as pen plotters, have a GPIB connection along with a serial or parallel port.

The GPIB board, however, must be programmed for the specific device which it will control. In most cases, drivers have to be built partially in a low level language to provide routines to communicate with the bus and to control the device. Many GPIB boards and devices have available drivers to add to existing code. These drivers often do not fit the user's requirements or are very difficult to interface with existing code because of language incompatibility problems. Many of the included drivers are written in BASIC programming language. The slow speed at which BASIC programs execute make these drivers inappropriate for many real-time applications.

HARDWARE INSTALLATION

The GPIB interface board used for this project was the Keithley MetraByte IE-488 General Interface I/O expansion board. It is designed to plug into an open slot inside a PC compatible computer. It fully complies with the IEEE-488 1978 standard.

The IE-488 board requires a free 16 kilobyte block of adapter memory to load a relocatable

12 kilobyte ROM interpreter and 4 kilobyte static RAM. The interpreter is a block of code that handles the initialization and protocol functions required to use the bus. The interpreter accepts all string coded commands and secondary commands in conventional high-level IEEE-488 command syntax, which will be discussed later. The interpreter must reside in a free portion of the PC's adapter memory. The memory address is selected by setting the appropriate D.I.P. switch on the GPIB board. The user must make sure that the set address does not conflict with any other devices connected to the computer. Figure 1 illustrates the hardware configuration of the Keithley Metrabyte IE-488 GPIB board.

Once an IEEE-488 command is sent to the ROM interpreter, the interpreter sends the necessary data out onto the GPIB through the I/O port address. This address must also be set on the board via a D.I.P. switch. This address performs the same function as the more common parallel and serial port addresses. The IE-488 Manual suggests selecting the base address at H300 or H310, although any other valid, non-conflicting address will work.

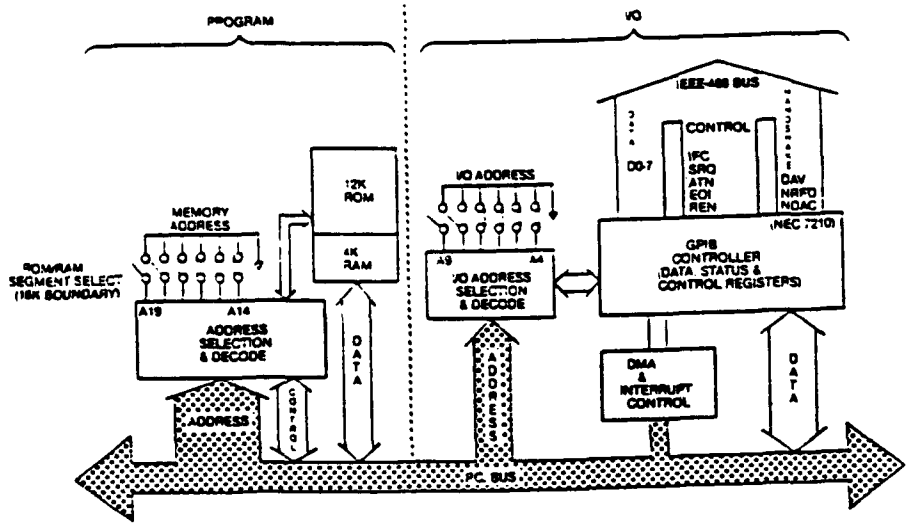


Figure 1. Hardware Diagram of Keithley Metrabyte IE-488 Board.
(Courtesy of Keithley Metrabyte)

If direct memory access (DMA) is going to be used, an appropriate interrupt and DMA channel must be specified by D.I.P. switches on the board. Care must also be taken with these to ensure that there are no conflicts with the operating system or another peripheral board.

The computer's technical reference manual should be consulted if there are any questions concerning the above parameter settings. The GPIB board manual will also assist in the setting of the appropriate parameters. The settings for the base address, memory address, interrupt vectors, and DMA channel should be noted so that the proper parameters can be set in the header file for the GPIB module. Although the discussion above pertained to the Keithley Metrabyte IE-488 GPIB board, a similar hardware setup is required for most other GPIB boards.

CONNECTING DEVICES USING THE GPIB

Devices may be connected to the GPIB in a linear configuration, a star configuration, or a combination of both. In a linear connection, the GPIB cable is connected from one device to the next, in a serial fashion (Figure 2). In a star configuration one end of all the GPIB connections are connected to one device (Figure 3). Each device is connected to the bus with either a GPIB or IEC-625-1 cable. The GPIB cable has a 24 pin connector. This connector is designed so that multiple cables can be connected "piggy-back" style. The pin diagram for this connector is shown in Appendix A.

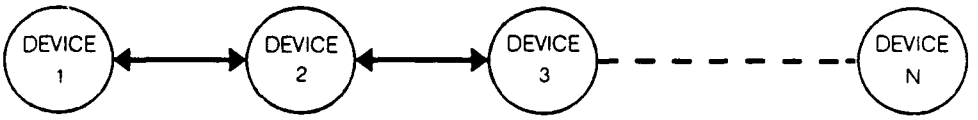


Figure 2. GPIB Linear Configuration.

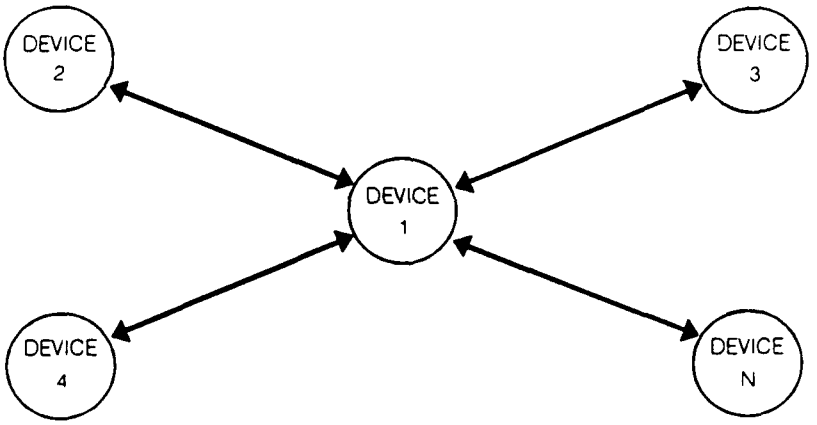


Figure 3. GPIB Star Configuration.

Devices that are interfaced to the GPIB can be classified into three categories:

1. Listeners - A listener is a device that receives commands and data via the bus.

2. Talkers - A talker is a device that sends data via the bus, to another device that is designated by the controller as a listener. Only one device on the bus is allowed to talk at any given time.

3. Controllers - The controller is a device that manages the interface bus by controlling the flow of commands and data to and from the other devices connected to the bus. Thus, the controller is both a listener and a talker. Only one active controller is allowed at any given time.

Many devices are designated as both talkers and listeners. For example, if only part of a waveform is of interest, the oscilloscope will first behave as a listener to receive the setup parameters for the window size. The scope will then become a talker as it passes the digitized signal to the controller.

The IEEE-488 specifications allow up to 15 devices to be connected to the bus, including the controller. Because of capacitive and inductive restrictions, the total length of the network should not exceed 20 meters. If a longer network is needed, an GPIB extender may be used. It is recommended that a device be connected for every 2 meters of cable, and that two-thirds of the devices be powered on. Appendix B lists some GPIB peripherals which may enhance the performance of an automated instrument system.

Each device connected to the bus must have a unique primary address. Many devices have a factory default address ranging from 0 to 30. However, this address can usually be changed by the user in different ways, depending upon the specific equipment in use. Some devices have switches or jumpers on the device which can be changed manually, while others may be changed from the instrument panel. Some devices will also have a secondary address, ranging from 96 - 126. This address will be used to control a specific function of that device. The secondary address is usually preset at the factory. It does not have to be unique, since the primary address must first be accessed. Not all devices support secondary addressing.

IEEE-488 INTERFACE COMMANDS

The commands that are interpreted by the GPIB board can be classified into two categories: universal commands and addressed commands. Below is a detailed description of each:

I. UNIVERSAL COMMANDS - A universal command will affect all of the devices that are connected, as long as they are capable of interpreting the command. There are five universal commands supported:

1. Serial Poll Enable (SPE) - Enables serial poll mode for all devices on the bus.

2. Serial Poll Disable (SPD) - Disables serial poll mode for all devices on the bus.

3. Parallel Poll Unconfigure (PPU) - All devices on the bus will be reset so that they are in an idle state. They will not be able to respond to a parallel poll.

4. Local Lockout (LLO) - Disables the front panel control for all devices on the bus.

5. Device Clear (DCL) - All devices on the bus will return to a predefined state. This predefined state is defined by the device.

II. ADDRESSED COMMANDS - An addressed command will affect only those devices which are addressed (designated as listeners). The controller selects which devices will receive the command. There are seven addressed commands supported:

1. Take Control (TCT) - The active controller will transfer bus control to the addressed device.

2. Parallel Poll Configure (PPC) - The addressed devices will be configured to respond to a parallel poll. This is done by assigning the 8 DIO lines of the bus for this purpose.

3. Select Device Clear (SDC) - The addressed devices will be reset to their predefined states. This predefined state is determined in the factory.

4. Go To Local (GTL) - The addressed devices will return to local control (they will be controlled by front panel operation).

5. Group Execute Trigger (GET) - The addressed devices will execute a pre-programmed action. This enables all of the selected devices on the bus to be triggered simultaneously.

6. UNTalk (UNT) - The current talker will be unaddressed. This command may not be necessary because when another device is designated as a talker, the previous talker is automatically unaddressed.

7. UNListen (UNL) - The bus is cleared of all listeners.

The Keithley Metrabyte IE-488 interpreter is capable of implementing all twelve of these commands. These commands, referred to as messages, are shown in Appendix D.

PROGRAMMING THE GPIB BOARD

The IEEE-488 standard does not mandate any particular programming language to be used with the GPIB board. However, there are certain formats that need to be followed. When establishing communication with the GPIB board, the following call statement is used:

CALL func(command, variable, flag, basadr)

where:

func is the name of the function that is to be called.

command is the string command that will determine what action the board will perform. This parameter will be decoded by the ROM interpreter on the GPIB board. The command must be sent to the board as a STRING in the following format:

"cmd dev1.sec, dev2.sec, ..., devn.sec[image]"

where:

cmd is the IEEE-488 command being sent.
devn is a user selectable device number.
sec is the devices secondary address (optional).
[image] is the image specifier or terminator (optional). The specifier is the variable field to which the acquired data is stored. The terminator allows the user to pick the type of string termination (EOI, carriage return, line feed, etc.). Details of these can be found in the GPIB board manual. The instrument driver described in this paper solely uses the *[$]* image specifier. This image specifier inputs the acquired data as a character string in memory. This allows for the maximum versatility and ease of programming since the character string can be easily transformed into an integer or float array if necessary. The included drivers use the default terminator: EOI.

variable is the command that is input to or output from the device. Data is transferred as specified by the image specifier and terminator. The format of this command is dependent upon the individual device. Consult the device's instruction manual for syntax requirements.

flag is a parameter that is used to report the status of an operation. If an error has occurred, this parameter will be set to the appropriate error number. The Keithley Metrabyte IE-488's error handling mechanism will identify 23 distinct types of messages grouped into three categories: data transfer, hardware, and format. The flag return codes for the IE-488 board are shown in Appendix C.

basadr is a variable that contains the address of the ROM interpreter.

Before sending any commands to a device, the GPIB board must be initialized. This is done by sending the appropriate *command* parameter to the GPIB board using the calling procedure discussed above. For example, when using the Keithley Metrabyte IE-488 the COMMAND string is:

"SYSCON MAD = dev, CIC = (0/1/2/3), NOB = (1/2), BAO=&Hdddd (BA1=&Hdddd)"

where:

MAD is the PC device address.
dev is the address of the device (0 - 30).
CIC is the controller in charge, 0=none, 1=board #1, 2=board #2.
NOB is the number of boards (1 or 2).
BAO is the base address for board #1.
BA1 is the base address for board #2.

The data string contained in *variable* is irrelevant when sending this command because no communication is being made with a device. For correct compilation of the code, however, a pointer to type string must be passed in the parameter list. After receiving this command, the board will be ready to accept further instructions.

SECTION 2
INSTRUMENT DRIVER INSTRUCTION MANUAL

GENERAL INFORMATION

The instrument driver code was written in Borland C++, Version 2.0. It is intended for a PC compatible 386 or 486 computer running MS-DOS 3.30 or higher, with at least one megabyte of memory, although it may run on other configurations. The software was modularly written and contains descriptive function names and comments describing the use of each function. There were two main modules coded: a GPIB module and a Tektronix module. The GPIB module contains the necessary functions to utilize the GPIB board with any device. It is in no way dependent upon any functions in the Tektronix module. The Tektronix module is designed to be layered on top of the GPIB module (see Figure 4). It accesses GPIB functions. To provide communication capabilities with additional devices, modules can be written similar to the Tektronix module. A small display module is provided to demonstrate the correct operation of the GPIB and Tektronix modules.

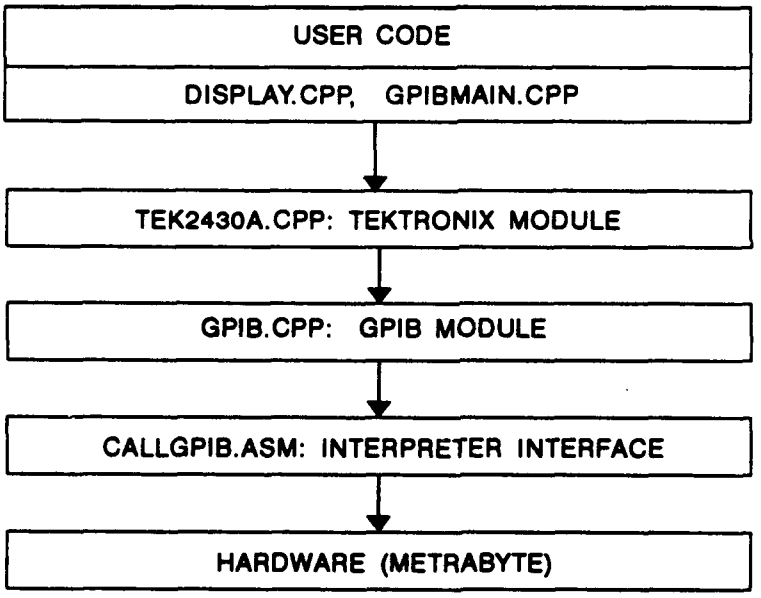


Figure 4. Layered Design of Driver Code.

To utilize these drivers, the user must include *gpib.h* and *tek2430A.h* in the program, compile *gpib.cpp* and *tek2430A.cpp*, assemble *callgpib.asm* and link all object files with the client code to produce an executable file. The user's code in the main program will then have access to all the GPIB and Tektronix functions described below. The display module, *display.cpp* and test program, *gpibmain.cpp* are included to demonstrate how the user can write code to access the functions provided by these drivers. The test program initializes the GPIB board and oscilloscope, sets the appropriate parameters on the oscilloscope, sends several queries to the oscilloscope, and then acquires and graphically displays the signal data.

Before compiling and linking the code, the user should set the default parameters for the board and the device. In the beginning of *gpib.h*, the user must define the interrupt request, DMA channel, PC device address, controller in charge, number of boards, I/O addresses, and ROM interpreter memory address of the GPIB board. In addition, the user must specify the length of the character array which will hold the commands to the GPIB board and the device. Below are the default settings of these parameters. The user should make any necessary changes:

```
#define DEF_IREQ 2
#define DEF_DMA 3
#define DEF_MAD 3
#define DEF_CIC 1
#define DEF_NOB 1
#define DEF_BA0 300
#define DEF_BA1 0
#define DEF_ROM_ADDR MK_FP(0xC800,0000)
#define IO_STR_LEN 80
```

In the Tektronix module the user must define the maximum number of points which can be acquired from the oscilloscope, and the device address of the oscilloscope. As with any GPIB device connected to the bus, the user must select a unique device number from the control panel of the oscilloscope. The default setting for this software is:

```
#define MAX_PTS 1024
#define DEF_GPIB_ADDR 15
```

GPIB MODULE

The GPIB module consists of the assembly language routine, *callgpib.asm*, along with the header file *gpib.h* and the implementation file *gpib.cpp*. The actual communication to the IEEE-488 interpreter is done through *callgpib.asm*. The addresses of the outgoing GPIB command structure, the device incoming/outgoing data structure, and the IEEE-488 ROM interpreter are parameters for this routine. This routine arranges these pointers along with an error flag pointer on the stack as illustrated in Figure 5.

It should also be noted that the command and data structure for the IEEE-488 interpreter have the following format:

```
class gpib_dat
{
  char length;
  char *dat_ptr;
        .
        .       // member functions
        .
};
```

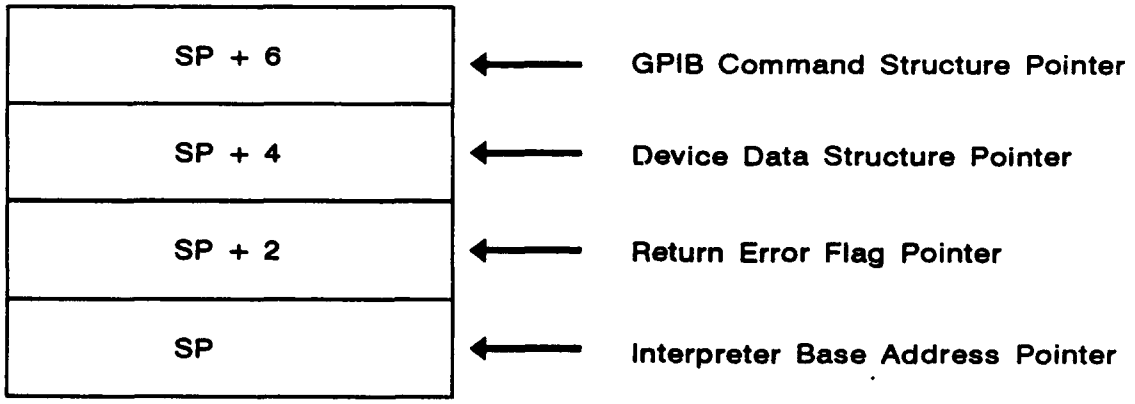


Figure 5. Location of Variables on the Stack.

This means that the command and data pointers on the stack will point to the length of the string. Directly following the length will be a pointer to the actual string somewhere else in memory. The form of the *callgpib* routine is as follows:

unsigned int flag = callgpib(gpib_cmd, dev_cmd_dat, DEF_ROM_ADDR);

where *gpib_cmd* is the command being sent to the GPIB board, and *dev_cmd_dat* is the command being sent to the device or data coming from the device. Both are of class *gpib_dat*. *DEF_ROM_ADDR* is the address of the IEEE-488 ROM interpreter. The C++ code incorporates constructors and destructors for class *gpib_dat* which dynamically allocates space for the data string.

Another class, *class gpib*, has the functions necessary for operation of the GPIB board. This class must be initialized with the appropriate parameters. If the default parameters accurately reflect the system configuration, these can be passed to the constructor as parameters. The main program illustrates the manner in which this is performed. The default parameters can also be over-ridden with subsequent calls to the *gpib* constructor by passing new values as parameters.

The following is a description of the service functions available once the GPIB module has been included in the user's code. Data access and modification methods are available but are not described because they are beyond the scope of this manual.

*gpib(int ireq_in, int dma_in, int mad_in, int cic_in, int nob_in, unsigned int ba0_in, unsigned int ba1_in, void far *rom_addr_in)* - This is the initializer for *class gpib*. The parameters represent the status of the user's system.

void get_params(int &ireq_out, int &dma_out, int &mad_out, int &cic_out, int &nob_out, unsigned int &ba0_out, unsigned int &ba1_out, void far &rom_addr_out)* - This function returns the data members in the *class gpib*.

gpib &copy(const gpib &gpib_in) - This function copies one gpib class to another. Both classes must be already created with the gpib constructor. The operator = is also overloaded to provide the same service.

unsigned int init_gpib() - This function initializes the GPIB board for use and returns the error code.

*unsigned int gpib_command(const char *gpib_cmd)* - This function sends *gpib_cmd* to the GPIB board and returns the error code.

*unsigned int device_command(const char *dev_cmd, const int dev_addr)* - This function sends *dev_cmd* to the device with device address *dev_addr* and returns the error code.

*unsigned int device_query(const int len, char *dev_dat, const int dev_addr)* - This function fills the empty string *dev_dat* of size *len* with data already waiting for it from device with address *dev_addr*. It returns the error code.

int error_handler(const unsigned int flag, const gpib_dat &last_cmd) - This function is usually called directly after *callgpib(...)* is called. When an error occurs, this function outputs an error description along with advice on how to correct the problem. The parameters are the *flag* which *callgpib(...)* returned and the last command, *last_cmd*, sent to the GPIB board. It returns 1 if the program should terminate and 0 if the program should continue to execute. The user is encouraged to change or add to the advice in this module as new solutions to errors are discovered.

The following functions are not necessary but are provided for the convenience of the user. They can all be implemented through the *gpib_command(char *gpib_cmd)* function. For example, to send the IEEE-488 command *"CLEAR 10,15"*, which will reset devices 10 and 15, either of the two methods can be used in client code to obtain the same result:

unsigned int flag = gpib_command("CLEAR 10,15"); or
unsigned int flag = clear("10,15");

Similarly, each of the following functions can be implemented by one of the above two ways. For clarity, each of the function names are identical to their Keithley Metrabyte IE-488 command syntax except for *abrt()*. *Abort* is a reserved word so it could not be used. A more detailed description of the following IE-488 commands can be found in the Keithley Metrabyte IE-488 GPIB board manual.

unsigned int abrt() - This function aborts current command and returns the error code.

*unsigned int clear(const char *devices)* - This function clears the selected *devices* and returns the error code.

*unsigned int config(const char *options)* - This function configures the GPIB board to the selected *options* and returns the error code.

*unsigned int enter(const char *device)* - This function inputs data from the selected *device* into a data string and returns the error code.

*unsigned int eoi(const char *device)* - This function sends a data byte on the selected *device* with EOI asserted and returns the error code.

*unsigned int local(const char *devices)* - This function sets the selected *devices* to local mode and returns the error code.

*unsigned int lockout(const char *devices)* - This function sets the selected *device* to local lockout and returns the error code.

*unsigned int output(const char *devices)* - This function outputs a string to the selected *devices* and returns the error code.

unsigned int parpol() - This function reads the status bit message for parallel poll devices and returns the error code.

*unsigned int pasctl(const char *device)* - This function passes control to the selected *device* and returns the error code.

*unsigned int ppconf(const char *device)* - This function sets up a parallel poll for the selected *device* and returns the error code.

*unsigned int ppuncf(const char *device)* - This function resets the parallel poll of the selected *device* and returns the error code.

*unsigned int remote(const char *devices)* - This function sets the selected *devices* into remote mode and returns the error code.

*unsigned int request(const char *device)* - This function requests service from active controller *device* and returns the error code.

unsigned int rxctl() - This function returns control of the bus to the PC and returns the error code.

*unsigned int status(const char *device)* - This function reads a serial polled *device* status byte and returns the error code.

*unsigned int syscon(const char *parameters)* - This function sets up the system configuration and initialization of the GPIB board according to *parameters* and returns the error code.

*unsigned int trigger(const char *devices)* - This function sends a trigger message to the selected *devices* and returns the error code.

TEKTRONIX MODULE

The Tektronix module was written to allow the user to control the Tektronix 2430A Digital Oscilloscope through the GPIB. There are only a few functions needed for this module. All string commands or queries may be sent through these functions. Other device modules can be coded using this as an example of how to interface with the GPIB module. All device commands and queries are passed through *tek_command(...)* and *tek_query(...)*. All the available functions are explained below:

tek2430A(const gpib &board_in, const int gpib_addr_in) - This is the initializer for *class tek2430A*. The parameters represent the initialized gpib board and the device's address.

unsigned int init_2430() - This function initializes the Tektronix oscilloscope for use and returns the error code.

*unsigned int tek_command(const char *cmd)* - This function sends the specified Tektronix command to the oscilloscope and returns the error code.

*char *s_tek_query(const char *query)* - This function sends the specified Tektronix query to the oscilloscope and returns the data in string format.

*float f_tek_query(const char *query)* - This function sends the specified Tektronix query to the oscilloscope and returns the data in floating point format.

*int i_tek_query(const char *query)* - This function sends the specified Tektronix query to the oscilloscope and returns the data in integer format.

*unsigned int capture_curve(const int num_pts, int *curve_out, const int i_channel)* - This function sends the curve query to the oscilloscope and stores the data from the specified *channel* into the specified integer array, *curve_out* of size *num_pts*. It returns the error code.

float tek_sample_rate() - This function returns the time between successive data points on the oscilloscope screen.

*void convert_to_volts(const int num_pts, float *f_curve, const int *i_curve)* - This function converts the integer curve array *i_curve* into voltages and stores it in the float array *f_curve*. The float curve must be of size *num_pts*.

DISPLAY MODULE AND TEST PROGRAM

A portion of a display module is provided to enhance demonstration of the instrument drivers. The header file *display.h* should be included in the program and *display.cpp* should be compiled and linked with the client code. This module contains the following function:

*int display_curve(const int num_pts, const float *dat, const float Y_AMP)* - This function causes the curve *dat* of *num_pts* to be displayed with a data amplification of *Y_AMP*. It returns 0 if executed properly and 1 if not.

The test program *gpibmain.cpp* initializes the GPIB board, the Tektronix oscilloscope, performs several operations, acquires a curve, and graphically displays it. This program illustrates the procedure for utilizing the modules correctly.

CONCLUSION

It has proven extremely beneficial to utilize the GPIB when interfacing laboratory instrumentation. The GPIB extends capabilities over what is normally available with serial and parallel ports. It provides a mechanism for both instrumentation control and data acquisition. Most laboratory equipment provides GPIB support or an option to add it.

The instrument drivers discussed in this paper were modularly written in a layered fashion to maximize portability and ease of understanding. Other modules can be plugged in to this software to expand its use to other GPIB devices. As is, this driver can be used to acquire data from the Tektronix 2430A digitizing oscilloscope. The data can be derived from any number of sources, such as ultrasonic transducer data, electronic sensor output, analog/digital circuit output, etc. These modules can be linked with other C or C++ code.

The instrument drivers worked very well with the test program. The oscilloscope was given its workout, the curve displayed on the oscilloscope was acquired by the computer into an array, and the graphics screen displayed the same curve shown on the oscilloscope. The source code for all the modules are included in Appendix E. Figure 6 shows a curve as it is displayed on the oscilloscope, and Figure 7 shows the same curve acquired through the GPIB and displayed on the computer with the display module. Figure 8 shows a block diagram of the hardware setup of this test and Figure 9 shows a photograph of the entire system.

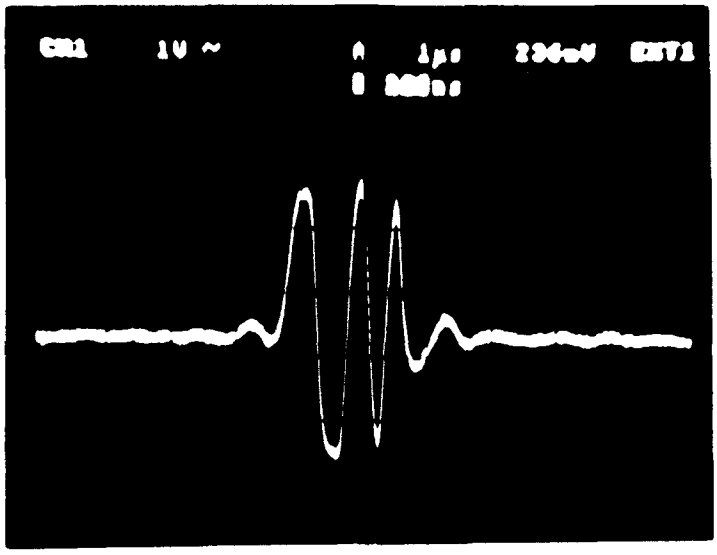


Figure 6. Curve Displayed on Oscilloscope.

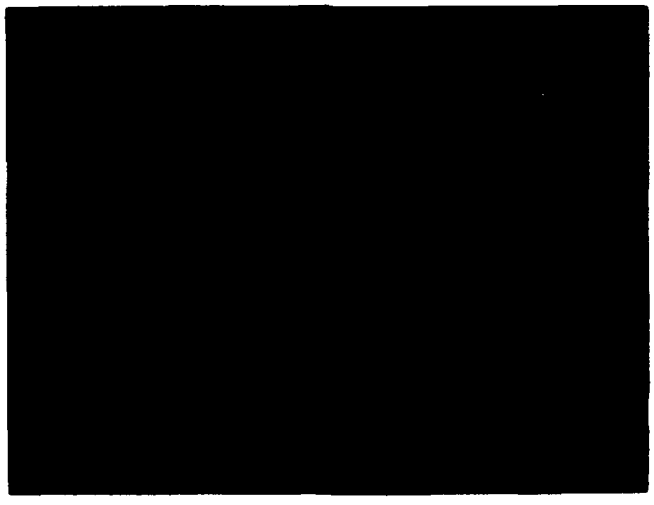


Figure 7. Curve Displayed on Computer Graphics Screen.

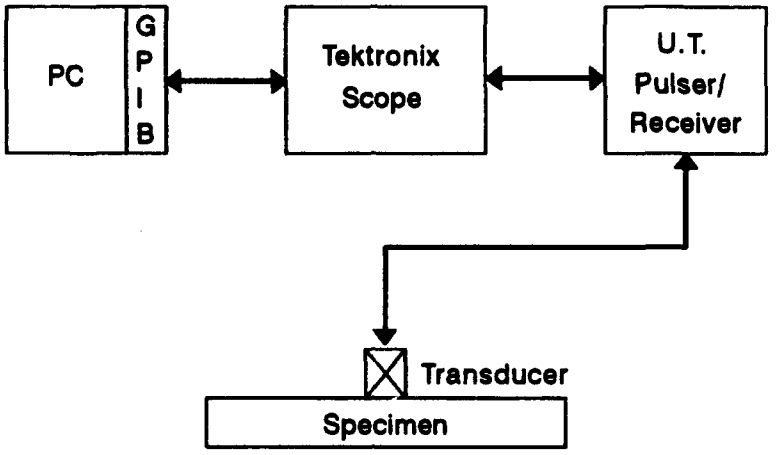


Figure 8. Block Diagram of Hardware Setup.

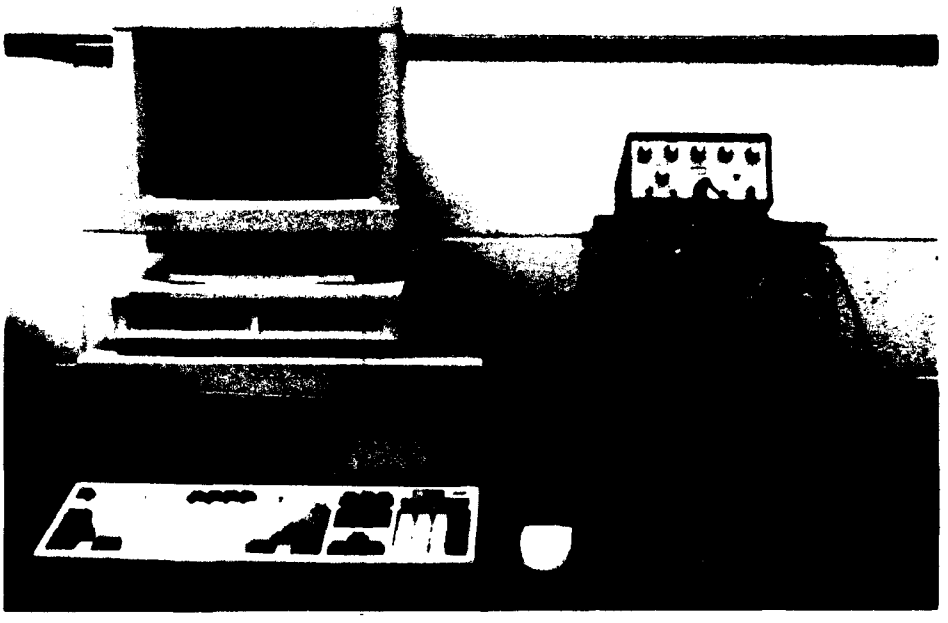


Figure 9. Photograph of Hardware Setup.

APPENDIX A

IEEE-488 PIN CONNECTIONS AND DIAGRAM

The pins can be classified into three groups (14):

1. DATA BUS

The pins labeled DIO1-DIO8 make up the data bus. These lines are used to pass data, commands, addresses, and status reports. The data is passed in a bit parallel, byte serial format.

2. DATA BYTE TRANSFER CONTROL

The three data byte transfer control lines are commonly referred to as the handshake bus. The purpose of handshaking is to ensure that all the information that is transferred on the bus is received by the proper device. The three lines are:

Not Ready For Data (NRFD) - This line is controlled by the devices that the controller has designated as a listener. If a listener is ready for data transfer, the devices line will be logically high. In order for the NRFD line to be high, all of the active listeners' NRFD lines must be high. Because of this, the NRFD line will operate at the speed of the slowest device.

Data Valid (DAV) - This line indicates that the data on the data bus is valid. The device that is designated by the controller as the talker controls the status of this line. This line will be set to low when the data is determined to be valid. This alerts the listeners on the bus that they can now accept the data. In order for this line to be activated low, the NRFD line must already be activated high.

No Data Accepted (NDAC) - This line is controlled by the devices that are designated by the controller as listeners. As each device receives the data from the bus, it sets its NDAC line to high. After all of the listeners have set their NDAC line to high, the cable NDAC line will be set high.

3. GENERAL INTERFACE MANAGEMENT LINES

The general interface management lines are used to control the bus and to report the status of select operations. The function of each line is described in the following:

Remote Enable (REN) - If this line is high, then the addressed device will be set to REMOTE operation. If this line is low, the addressed device will be controlled by the operator.

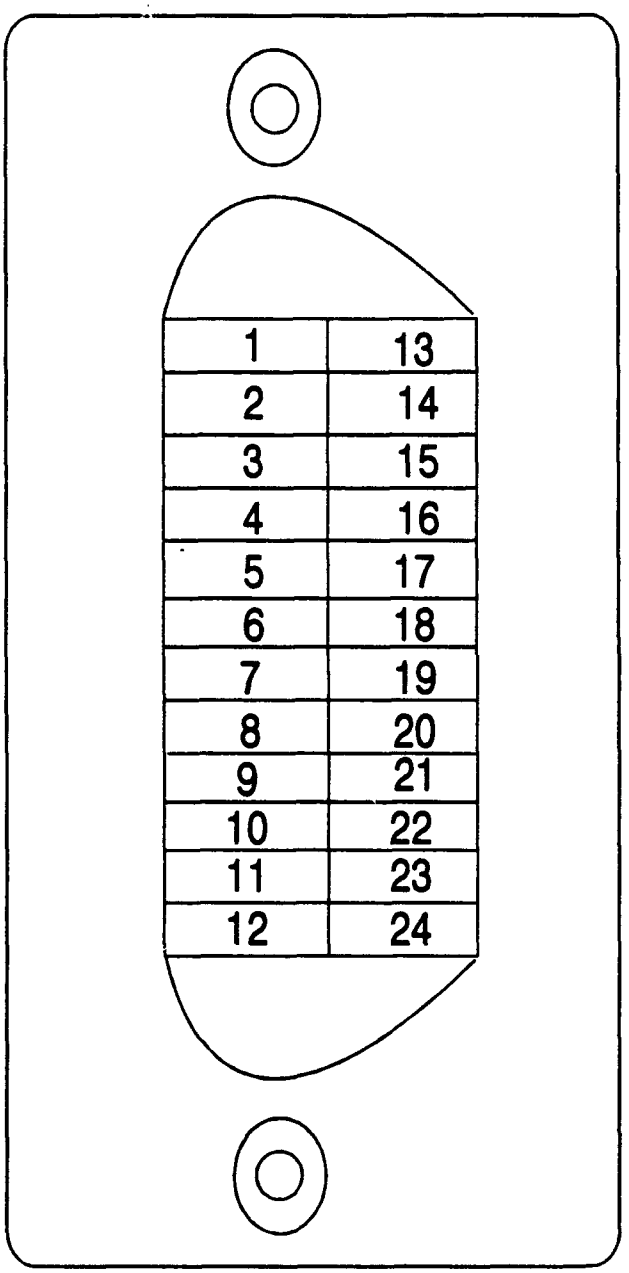
Attention (ATN) - This line will describe the information that is on the bus. If this line is high, then the information is considered to be data. All of the devices designated by the controller as listeners will receive data from the device designated as a talker. If this line is low, then the information on the bus is considered to be a command. All devices on the bus will receive commands.

End Or Identify (EOI) - This line works in conjunction with the ATN line. If both the EOI line and the ATN line are high, then the controller will initiate a parallel poll of the status of each device. If ATN is low and EOI is high, then the last byte in a data message has been located.

Interface Clear (IFC) - If this line is high, then the bus will be set to an idle state. This most often occurs when there is a problem.

Service Request (SRQ) - If this line is high, it indicates that the device that activated it requests attention. As soon as the controller is free, it will let the calling device perform its operation.

IEEE-488 PIN DIAGRAM (5)



D101	1	13	D105
D102	2	14	DI06
D103	3	15	DI07
EOI	4	16	DI08
DAV	5	17	REN
NRFD	6	18	GND (TW PAIR W/DAV)
NDAC	7	19	GND (TW PAIR W/NRFD)
IFC	8	20	GND (TW PAIR W/NDAC)
SRQ	9	21	GND (TW PAIR W/IFC)
ATN	10	22	GND (TW PAIR W/SRQ)
SHIELD	11	23	GND (TW PAIR W/ATN)
	12	24	SIGNAL GROUND

APPENDIX B

SUPPLEMENTAL GPIB HARDWARE (4)

As the GPIB interfacing system became more widely accepted, more tools were developed to enhance its performance. The hardware listed here is not mandatory for a system that utilizes the GPIB, but may enhance system performance.

1. GPIB MODEM

This modem will allow a device to be connected via a telephone line. However, because modems transfer data bits serially, the transfer will be slow. Thus, a modem is not advisable for a system that will make a large number of transfers. A major advantage of the use of a modem is that the operator can be physically away from the system, with operation accomplished via telephone.

2. SERIAL TO GPIB CONVERTER

Currently, most computers come with a standard serial (RS-232) port, but not a GPIB (IEEE-488). This converter will allow the serial port to be converted into a GPIB. An advantage of this is that the converter eliminates the cable length restrictions placed on normal GPIB systems. Also, any device with a serial port can be converted into a GPIB instrument. Devices that are connected to this converter can be listeners or talkers, since the RS-232 is bi-directional. There will, however, be a decrease in speed performance if this hardware is employed.

3. PARALLEL TO GPIB CONVERTER

This converter allows any unit with a parallel port to be used as a GPIB instrument. A limitation to this device is that it must be used with listen only devices, since the parallel port is used for output only.

4. EXPANDER

An expander is a device that will allow more than 15 devices to be connected to the GPIB. There is, however, still a limit of 31 total devices which can be connected at one time. This occurs because each device on the bus must have a unique address ranging from 0 to 30.

5. EXTENDER

An extender is a device that will allow devices to be distanced more than the recommended 2 meters. This can also be used to achieve greater electrical isolation and less data transmission errors due to electrical interference.

6. DATA BUFFER

This device is used to improve the efficiency of the system. Usually the systems performance is based on the performance of the slowest device. This problem is overcome by isolating the slowest device on the bus through the use of a data buffer. This buffer will be connected to both the slowest device and the interfaced system. The buffer will allow the slow device to perform its operation, while the rest of the devices continue to perform theirs. An example of this is a system that plots output. It often takes quite some time to accomplish this. Without a data buffer, the whole system will be at a standstill until the plot is complete. With a buffer, the other devices can perform their operations at the same time as the plotter, leading to significant performance improvements.

APPENDIX C

KEITHLEY METRABYTE IE-488 ERROR FLAG CODES (8)

The error codes listed will be returned in the FLAG parameter of the GPIB call statement. These codes can also be found in the Keithley Metrabyte IE-488 Manual.

DATA TRANSFER ERRORS

- &H0000 = Successful Transfer
- &H0020 = No Input EOI or Line Feed
- &H0030 = Device Timeout
- &H0040 = Reserved
- &H0050 = DMA Channel Busy
- &H0060 = GPIB Busy

HARDWARE ERRORS

- &H0100 = Hardware Failure
- &H0200 = Time Out on Data Transfer
- &H0300 = Device Not Active Controller
- &H0400 = IBM PC Active Controller
- &H0500 = System Not Initialized
- &H0600 = Configuration Error

FORMAT ERRORS

- &H1000 = Undefined Command
- &H1100 = Syntax Error in Command Line
- &H2000 = Undefined Image
- &H3000 = Device Range Error
- &H3100 = Too Many Devices
- &H3200 = Talker/Listener Conflict
- &H4000 = Command/Data Out of Range
- &H5000 = Command Requires Device
- &H6000 = Undefined Device Code
- &H7000 = Input Array not Initialized
- &H9000 = IBM Must be Talker or Listener

APPENDIX D

KEITHLEY METRABYTE IE-488 INTERFACE COMMANDS (8)

The interpreter on the Keithley MetraByte IE-488 Interface Board implements all twelve standard IEEE-488 messages. Sending the following messages to the IE-488 Board will effect the appropriate pins in Appendix A.

1. DATA MESSAGE - Data information sent from the talker device to the listener devices.

2. TRIGGER MESSAGE - Causes the listener devices to perform a specific action.

3. CLEAR MESSAGE - Causes addressed devices to return to their predefined state.

4. REMOTE MESSAGE - Causes addressed devices to switch from panel to remote program control.

5. LOCAL MESSAGE - Causes remote message to be cleared from addressed device.

6. LOCAL LOCKOUT MESSAGE - Prevents operator from manually returning to local control from the panel.

7. CLEAR LOCKOUT MESSAGE - Clears all devices on the GPIB and sets to local mode.

8. SERVICE REQUEST MESSAGE - A device sends this to request service from the controller. This is cleared by sending the device's Status Byte Message when it no longer requires service.

9. STATUS BYTE MESSAGE - A data byte which represents the status of the device on the bus. Bit 6 is set if the device is sent a Service Request Message, and the remaining bits are specific to the selected device.

10. STATUS BIT MESSAGE - A byte which represents the operational conditions of a group of devices on the bus. Each bit represents a device on the bus. This is a response from Parallel Poll Operation.

11. PASS CONTROL MESSAGE - Allows transfer of bus management duties to another device on the bus.

12. ABORT MESSAGE - System controller takes unconditional control of the device from the active controller. This message terminates communications with the bus and sends a Clear All Message.

APPENDIX E

SOURCE CODE

```
.model   MEDIUM
;*****************************************************************************
;*                                CALLGPIB.ASM                               *
;*****************************************************************************
;* Evolved from work of: Jeffrey Gruber, Hollis Graves, Andrew Klumpp,       *
;*                       Patrick Sincebaugh, and Bradley M. Taber III        *
;* For: U.S. Army Materials Technology Laboratory                            *
;*      Materials Testing and Evaluation Branch                              *
;* Date: 31 March 1992                                                       *
;*****************************************************************************
;* This routine passes command and data strings through the GPIB board       *
;*****************************************************************************

;---------------------------------------------------------------------------
                .DATA
;---------------------------------------------------------------------------

;Equate Statements
gpib_rom_addr   EQU     DWORD PTR [bp+10]       ;GPIB ROM Address
data            EQU     WORD PTR  [bp+8]        ;device command or data
command         EQU     WORD PTR  [bp+6]        ;GPIB command
flag            EQU     WORD PTR  [bp-6]        ;error flag
base            EQU     WORD PTR  [bp-8]        ;address of board = 0

;---------------------------------------------------------------------------
                .CODE
;---------------------------------------------------------------------------

                PUBLIC _callgpib
_callgpib       PROC

                push    bp              ;save base pointer
                mov     bp, sp          ;set up stack pointer
                push    di              ;save data index
                push    si              ;save source index
                sub     sp, 4           ;reserve space for local variables
                                        ;flag and base

                ;for debugging purposes, omit ';' and monitor ax
                ;mov    ax, command
                ;mov    ax, data
                ;mov    ax, [bp+10]     ;gpib_rom_addr
                ;mov    ax, [bp+12]     ;gpib_rom_addr

                xor     ax, ax          ;set ax to 0
                mov     flag, ax        ;set flag to 0
                mov     base, ax        ;set base to 0

                push    command         ;push command
                push    data            ;push data
                mov     ax, bp          ;get pointer to local variables
                sub     ax, 6           ;adjust to point to flag
                push    ax              ;push flag
                sub     ax, 2           ;adjust to point to base
                push    ax              ;push base

                call    gpib_rom_addr   ;call the GPIB ROM

                ; After return from GPIB board, the pointers (command, data,
                ; flag, base) are off of the stack

                pop     ax              ;get flag
                pop     ax              ;get flag, return it in ax register

                pop     si              ;recover source index
                pop     di              ;recover data index
                pop     bp              ;recover base pointer
                ret                     ;return to calling function
_callgpib       ENDP
                END
;---------------------------------------------------------------------------
```

```
/*****************************************************************************
 *                                  GPIB.H                                   *
 *****************************************************************************
 * Written by: Bradley M. Taber III                                          *
 * For: U.S. Army Materials Technology Laboratory                            *
 *      Materials Testing and Evaluation Branch                              *
 * Date: 6 April 1992                                                        *
 *****************************************************************************
 * Include file for use with GPIB communications board                       *
 *****************************************************************************/

#ifndef GPIB_H
#define GPIB_H

#include <fstream.h>

#define DEF_IREQ 2                          //default GPIB board settings
#define DEF_DMA 3
#define DEF_MAD 3
#define DEF_CIC 1
#define DEF_NOB 1
#define DEF_BA0 300
#define DEF_BA1 0
#define DEF_ROM_ADDR MK_FP(0xC800,0000)

#define IO_STR_LEN 80                       //default command length

/*--------------------------------------------------------------------------*/

//This is the data sructure for command and data I/O with the GPIB board
class gpib_dat
{
   private:
   //Data Members
      char length;                                  //length of output string
      char *dat_str;                                //pointer to string

   public:
   //Constructors / Destructor
      gpib_dat(const char *dat_str_in);             //constructors - commands
      gpib_dat(const int size_in);                  //             - data in
     ~gpib_dat();                                   //destructor

   //Access / Modify
      void set_length(const char length_in);
      char get_length() const;
      void set_dat(const char *dat_str_in);
      void get_dat(char *dat_str_out) const;
      char get_dat_ele(const int index);

   //Output
      friend ostream &operator<<(ostream &out_stream,
         const gpib_dat &gpib_dat_in);
};

/*--------------------------------------------------------------------------*/

//This class represents the GPIB board and all the necessary parameters
class gpib
{
   private:
   //Data Members
      int ireq;
      int dma;
      int mad;
      int cic;
      int nob;
      unsigned int ba0;
      unsigned int ba1;
      void far *rom_addr;
```

```
   public:
   //Constructors
      gpib();
      gpib(int ireq_in, int dma_in, int mad_in, int cic_in, int nob_in,
         unsigned int ba0_in, unsigned int ba1_in, void far *rom_addr_in);

   //Access / Modify
      void get_params(int &ireq_out, int &dma_out, int &mad_out,
         int &cic_out, int &nob_out, unsigned int &ba0_out,
         unsigned int &ba1_out, void far* &rom_addr_out);
      gpib &gpib::copy(const gpib &gpib_in);
      gpib &gpib::operator = (const gpib &gpib_in);

   //GPIB board functions
      unsigned int init_gpib();
      unsigned int gpib_command(const char *gpib_cmd);
      unsigned int device_command(const char *dev_cmd, const int dev_addr);
      unsigned int device_query(const int len, char *dev_dat,
         const int dev_addr);
      int error_handler(const unsigned int flag_in, const gpib_dat &last_cmd);
      unsigned int abrt();
      unsigned int clear(const char *devices);
      unsigned int config(const char *options);
      unsigned int enter(const char *device);
      unsigned int eoi(const char *device);
      unsigned int local(const char *devices);
      unsigned int lockout(const char *devices);
      unsigned int output(const char *devices);
      unsigned int parpol();
      unsigned int pasctl(const char *devices);
      unsigned int ppconf(const char *device);
      unsigned int ppuncf(const char *device);
      unsigned int remote(const char *devices);
      unsigned int request(const char *device);
      unsigned int rxctl();
      unsigned int status(const char *device);
      unsigned int syscon(const char *parameters);
      unsigned int trigger(const char *devices);
};

//Assembly language routine for communication with GPIB board
extern "C" unsigned int callgpib(const gpib_dat *cmd_out, gpib_dat *data_in,
   const void far *rom_addr);

#endif   //GPIB_H
```

```
/*************************************************************************
 *                                GPIB.CPP                               *
 *************************************************************************
 * Written by: Bradley M. Taber III                                      *
 * For: U.S. Army Materials Technology Laboratory                        *
 *      Materials Testing and Evaluation Branch                          *
 * Date: 6 April 1992                                                    *
 *************************************************************************
 * Implementation file for GPIB.H                                        *
 *************************************************************************/

#include <assert.h>
#include <conio.h>
#include <dos.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

#include "gpib.h"

/*-----------------------------------------------------------------------*/
/*                          class gpib_dat                               */
/*-----------------------------------------------------------------------*/

//Constructor  -- for string output only
inline gpib_dat::gpib_dat(const char *dat_str_in)
{
   length = strlen(dat_str_in);
   dat_str = new char[length+1];
   assert(dat_str != 0);
   strcpy(dat_str, dat_str_in);
}

/*-----------------------------------------------------------------------*/

//Constructor  -- for data input only
gpib_dat::gpib_dat(const int size_in)
{
   dat_str = new char[size_in];
   assert(dat_str != 0);
   for (register int c=0; c<size_in; c++)
      dat_str[c] = '\0';                        //NULL
   length = 0;                                  //not used for input
}

/*-----------------------------------------------------------------------*/

//Destructor
inline gpib_dat::~gpib_dat()
{
   delete dat_str;
}

/*-----------------------------------------------------------------------*/

//Set length
inline void gpib_dat::set_length(const char length_in)
{
   length = length_in;
}

/*-----------------------------------------------------------------------*/

//Get length
inline char gpib_dat::get_length() const
{
   return length;
}

/*-----------------------------------------------------------------------*/
```

```
//Set dat_str
inline void gpib_dat::set_dat(const char *dat_str_in)
{
   strcpy(dat_str, dat_str_in);
}

/*-------------------------------------------------------------------------*/

//Get dat_str
inline void gpib_dat::get_dat(char *dat_str_out) const
{
   strcpy(dat_str_out, dat_str);
}

/*-------------------------------------------------------------------------*/

//Get dat_str element
inline char gpib_dat::get_dat_ele(const int index)
{
   return dat_str[index];
}

/*-------------------------------------------------------------------------*/

//Output -- for output strings only
ostream &operator<<(ostream &out_stream, const gpib_dat &gpib_dat_in)
{
   char len = gpib_dat_in.get_length();
   char *str = new char[len+1];
   assert(str != 0);
   gpib_dat_in.get_dat(str);
   cout << "!" << str << "' (" << int(len) << ")";
   delete str;
   return out_stream;
}

/*-------------------------------------------------------------------------*/
/*                              class gpib                                 */
/*-------------------------------------------------------------------------*/

//Constructor
gpib::gpib()
{
   ireq = 0;
   dma  = 0;
   mad  = 0;
   cic  = 0;
   nob  = 0;
   ba0  = 0;
   ba1  = 0;
   rom_addr = MK_FP(0x0000,0000);
}

/*-------------------------------------------------------------------------*/

//Constructor
gpib::gpib(int ireq_in, int dma_in, int mad_in, int cic_in, int nob_in,
   unsigned int ba0_in, unsigned int ba1_in, void far *rom_addr_in)
{
   ireq = ireq_in;
   dma  = dma_in;
   mad  = mad_in;
   cic  = cic_in;
   nob  = nob_in;
   ba0  = ba0_in;
   ba1  = ba1_in;
   rom_addr = rom_addr_in;
}

/*-------------------------------------------------------------------------*/
```

```
//Get GPIB Parameters
inline void gpib::get_params(int &ireq_out, int &dma_out, int &mad_out,
   int &cic_out, int &nob_out, unsigned int &ba0_out, unsigned int &ba1_out,
   void far* &rom_addr_out)
{
   ireq_out = ireq;
   dma_out  = dma;
   mad_out  = mad;
   cic_out  = cic;
   nob_out  = nob;
   ba0_out  = ba0;
   ba1_out  = ba1;
   rom_addr_out = rom_addr;
}

/*------------------------------------------------------------------------*/

//Copy gpib
inline gpib &gpib::copy(const gpib &gpib_in)
{
   ireq = gpib_in.ireq;
   dma  = gpib_in.dma;
   mad  = gpib_in.mad;
   cic  = gpib_in.cic;
   nob  = gpib_in.nob;
   ba0  = gpib_in.ba0;
   ba1  = gpib_in.ba1;
   rom_addr = gpib_in.rom_addr;
   return *this;
}

/*------------------------------------------------------------------------*/

//Overload operator =
gpib &gpib::operator = (const gpib &gpib_in)
{
   if (this != &gpib_in) return copy(gpib_in);
}

/*------------------------------------------------------------------------*/

//Initialize GPIB board
unsigned int gpib::init_gpib()
{
   char cmd[IO_STR_LEN], temp[IO_STR_LEN];

   strcpy(cmd, "SYSCON MAD=\0");       //system configuration, must be
   itoa(mad, temp, 10);                //sent before any other commands
   strcat(cmd, temp);

   strcat(cmd, ",CIC=\0");
   itoa(cic, temp, 10);
   strcat(cmd, temp);

   strcat(cmd, ",NOB=\0");
   itoa(nob, temp, 10);
   strcat(cmd, temp);

   strcat(cmd, ",BA0=&H\0");
   itoa(ba0, temp, 10);
   strcat(cmd, temp);

   if (nob == 2)
   {
      strcat(cmd, ",BA1=&H\0");
      itoa(ba1, temp, 10);
      strcat(cmd, temp);
   }
```

```
   gpib_command(cmd);
   gpib_command("TIMEOUT\0");
   gpib_command("ABORT\0");
   unsigned int flag = gpib_command("CLEAR\0");
   return flag;
}

/*-------------------------------------------------------------------------*/

//Send command to GPIB board
unsigned int gpib::gpib_command(const char *gpib_cmd)
{
   gpib_dat cmd_out(gpib_cmd);
   gpib_dat data_out(1);                    //data_out doesn't hold anything
                                             //but pointer is still necessary

   unsigned int flag = callgpib(&cmd_out, &data_out, rom_addr);
   if (flag) error_handler(flag, cmd_out);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Send command to device
unsigned int gpib::device_command(const char *dev_cmd,
   const int dev_addr)
{
   char gpib_cmd[IO_STR_LEN] = "OUTPUT \0";          //must be sent for
   char dev_addr_str[3];
   itoa(dev_addr, dev_addr_str, 10);
   strcat(gpib_cmd, dev_addr_str);                   //device command
   strcat(gpib_cmd, "[$]\0");

   gpib_dat cmd_out(gpib_cmd), data_out(dev_cmd);
   unsigned int flag = callgpib(&cmd_out, &data_out, rom_addr);
   if (flag) error_handler(flag, cmd_out);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Get data from device
unsigned int gpib::device_query(const int len, char* dev_dat,
   const int dev_addr)
{
   char gpib_cmd[IO_STR_LEN] = "ENTER \0";           //must be sent for
   char dev_addr_str[3];
   itoa(dev_addr, dev_addr_str, 10);
   strcat(gpib_cmd, dev_addr_str);                    //for device input
   strcat(gpib_cmd, "[$]\0");

   gpib_dat cmd_out(gpib_cmd), data_in(len);
   unsigned int flag = callgpib(&cmd_out, &data_in, rom_addr);
   if (flag) error_handler(flag, cmd_out);

   for (register int c=0; c<len; c++)
      dev_dat[c] = data_in.get_dat_ele(c);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Error handler
/* The user is encouraged to change or add to the advise when solutions
   to errors are discovered.  This function returns 0 if the program can
   continue and 1 if the program should terminate.  It is up to the user
   to implement a graceful termination if necessary.                    */
```

```
int gpib::error_handler(const unsigned int flag_in, const gpib_dat &last_cmd)
{
   if (flag_in == 0) return 0;                    //get out if no error
   int flag;

   sound(500);  delay(100);                       //sound alarm
   sound(1000); delay(100);
   sound(500);  delay(100);
   nosound();

   cerr << "\n*** GPIB ERROR ***\n";
   cerr << "IN COMMAND " << last_cmd << '\n';
   cerr << "ERROR CODE " << flag_in << '\n';
   switch (flag_in)
   {
      case 0x0020:
         cerr << "NO INPUT EOI or LINE FEED\n";
         cerr << "Advise: Check image terminator\n";
         flag = 0;
         break;
      case 0x0030:
         cerr << "DEVICE TIME OUT\n";
         cerr << "Advise: consult manual\n";
         flag = 0;
         break;
      case 0x0040:
         cerr << "RESERVED\n";
         cerr << "Advise: consult manual\n";
         flag = 0;
         break;
      case 0x0050:
         cerr << "DMA CHANNEL BUSY\n";
         cerr << "Advise: try again when cleared\n";
         flag = 0;
         break;
      case 0x0060:
         cerr << "GPIB BUSY\n";
         cerr << "Advise: consult manual\n";
         flag = 0;
         break;
      case 0x0100:
         cerr << "HARDWARE FAILURE\n";
         cerr << "Advise: check GPIB I/O address\n";
         flag = 1;
         break;
      case 0x0200:
         cerr << "TIME OUT ON DATA TRANSFER\n";
         cerr << "Advise: check GPIB cable connection, GPIB address, and\n";
         cerr << "        make sure device is on\n";
         flag = 0;
         break;
      case 0x0300:
         cerr << "DEVICE NOT ACTIVE CONTROLLER\n";
         cerr << "Advise: check device number or make device active controller\n";
         flag = 0;
         break;
      case 0x0400:
         cerr << "IBM-PC ACTIVE CONTROLLER\n";
         cerr << "Advise: consult manual\n";
         flag = 0;
         break;
      case 0x0500:
         cerr << "SYSTEM NOT INITIALIZED\n";
         cerr << "Advise: execute function init_gpib() to initialize\n";
         flag = 1;
         break;
      case 0x0600:
         cerr << "CONFIGURATION ERROR\n";
         cerr << "Advise: check SYSCON... in init_gpib()\n";
         flag = 1;
         break;
```

```
        case 0x1000:
            cerr << "UNDEFINED COMMAND\n";
            cerr << "Advise: Use available command\n";
            flag = 0;
            break;
        case 0x1100:
            cerr << "SYNTAX ERROR IN COMMAND LINE\n";
            cerr << "Advise: check command syntax\n";
            flag = 0;
            break;
        case 0x2000:
            cerr << "UNDEFINED IMAGE\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
        case 0x3000:
            cerr << "DEVICE RANGE ERROR\n";
            cerr << "Advise: make sure device numbers are within specified range\n";
            flag = 0;
            break;
        case 0x3100:
            cerr << "TOO MANY DEVICES\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
        case 0x3200:
            cerr << "TALKER/LISTENER CONFLICT\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
        case 0x4000:
            cerr << "COMMAND/DATA OUT OF RANGE\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
        case 0x5000:
            cerr << "COMMAND REQUIRES DEVICE\n";
            cerr << "Advise: execute command on specific device number(s)\n";
            flag = 0;
            break;
        case 0x6000:
            cerr << "UNDEFINED DEVICE CODE\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
        case 0x7000:
            cerr << "INPUT ARRAY NOT INITIALIZED\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
        case 0x9000:
            cerr << "IBM MUST BE TALKER or LISTENER\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
        default:
            cerr << "UNDEFINED ERROR CODE\n";
            cerr << "Advise: consult manual\n";
            flag = 0;
            break;
    }
    if (flag == 0) cerr << "Program can continue\n";
    else cerr << "Program should terminate\n";

    cerr << "HIT ANY KEY\n";
    getch();

    return flag;
}
```

```
/*-------------------------------------------------------------------------*
 * The following functions are not necessary for this driver but are       *
 * provided as a convenience to the user.                                  *
 *-------------------------------------------------------------------------*/

//Abort
inline unsigned int gpib::abrt()
{
   unsigned int flag = gpib_command("ABORT\0");
   return flag;
}

/*-------------------------------------------------------------------------*/

//Clear devices
inline unsigned int gpib::clear(const char *devices)
{
   char cmd[IO_STR_LEN] = "CLEAR \0";
   strcat(cmd, devices);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Config GPIB
inline unsigned int gpib::config(const char *options)
{
   char cmd[IO_STR_LEN] = "CONFIG \0";
   strcat(cmd, options);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Input GPIB data
inline unsigned int gpib::enter(const char *device)
{
   char cmd[IO_STR_LEN] = "ENTER \0";
   strcat(cmd, device);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Send data byte with EOI asserted
inline unsigned int gpib::eoi(const char *device)
{
   char cmd[IO_STR_LEN] = "EOI \0";
   strcat(cmd, device);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Set selected devices
inline unsigned int gpib::local(const char *devices)
{
   char cmd[IO_STR_LEN] = "LOCAL \0";
   strcat(cmd, devices);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/
```

```
//Lockout selected devices
inline unsigned int gpib::lockout(const char *devices)
{
   char cmd[IO_STR_LEN] = "LOCKOUT \0";
   strcat(cmd, devices);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Output string to selected devices
inline unsigned int gpib::output(const char *devices)
{
   char cmd[IO_STR_LEN] = "OUTPUT \0";
   strcat(cmd, devices);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Reads status bit message for parallel poll devices
inline unsigned int gpib::parpol()
{
   unsigned int flag = gpib_command("PARPOL\0");
   return flag;
}

/*-------------------------------------------------------------------------*/

//Pass control to selected device
inline unsigned int gpib::pasctl(const char *device)
{
   char cmd[IO_STR_LEN] = "PASCTL \0";
   strcat(cmd, device);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Set up parallel poll
inline unsigned int gpib::ppconf(const char *device)
{
   char cmd[IO_STR_LEN] = "PPCONF \0";
   strcat(cmd, device);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Resets parallel poll
inline unsigned int gpib::ppuncf(const char *device)
{
   char cmd[IO_STR_LEN] = "PPUNCF \0";
   strcat(cmd, device);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Sets selected devices to go into remote
inline unsigned int gpib::remote(const char *devices)
{
   char cmd[IO_STR_LEN] = "REMOTE \0";
   strcat(cmd, devices);
   unsigned int flag = gpib_command(cmd);
   return flag;
}
```

```
/*-------------------------------------------------------------------------*/

//Requests service from active controller
inline unsigned int gpib::request(const char *device)
{
   char cmd[IO_STR_LEN] = "REQUEST \0";
   strcat(cmd, device);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Returns control of the bus to the PC
inline unsigned int gpib::rxctl()
{
   unsigned int flag = gpib_command("RXCTL\0");
   return flag;
}

/*-------------------------------------------------------------------------*/

//Reads serial polled device status byte
inline unsigned int gpib::status(const char *device)
{
   char cmd[IO_STR_LEN] = "STATUS \0";
   strcat(cmd, device);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Sets up system configuration and initialization of GPIB
inline unsigned int gpib::syscon(const char *parameters)
{
   char cmd[IO_STR_LEN] = "SYSCON \0";
   strcat(cmd, parameters);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Sends trigger message to selected devices
inline unsigned int gpib::trigger(const char *devices)
{
   char cmd[IO_STR_LEN] = "TRIGGER \0";
   strcat(cmd, devices);
   unsigned int flag = gpib_command(cmd);
   return flag;
}

/*-------------------------------------------------------------------------*/
```

```
/*****************************************************************************
 *                              TEK2430A.H                                   *
 *****************************************************************************
 * Written by: Bradley M. Taber III                                          *
 * For: U.S. Army Materials Technology Laboratory                            *
 *      Materials Testing and Evaluation Branch                              *
 * Date: 6 April 1992                                                        *
 *****************************************************************************
 * Include file for use with Tektronix 2430A Oscilloscope                    *
 *****************************************************************************/

#ifndef TEK2430A_H
#define TEK2430A_H

#define MAX_PTS 1024                        //Maximum number of points

#define DEF_GPIB_ADDR 15                    //Tektronix GPIB address,
                                              //range(0-30), make unique
                                              //max data points for curve

class tek2430A
{
   private:
   //Data Members
      gpib board;
      int gpib_addr;

   public:
   //Constructor
      tek2430A(const gpib &board_in, const int gpib_addr_in = DEF_GPIB_ADDR);

   //Initialize Tek 2430A Scope
      unsigned int tek_init();

   //Access / Modify
      void set_board(const gpib &board_in);
      gpib get_board();
      void set_gpib_addr(const int gpib_addr_in);
      int get_gpib_addr();

   //Device Input / Output
      unsigned int tek_command(const char *cmd);
      char *s_tek_query(const char *query);
      float f_tek_query(const char *query);
      int i_tek_query(const char *query);
      unsigned int capture_curve(const int num_pts, int *curve_out,
         const int i_channel);
      float tek_sample_rate();

   //Conversion
      void convert_to_volts(const int num_pts, float *f_curve,
         const int *i_curve);
};

#endif   //TEK2430A_H
```

```
/*************************************************************************
 *                              TEK2430A.CPP                             *
 *************************************************************************
 * Written by: Bradley M. Taber III                                      *
 * For: U.S. Army Materials Technology Laboratory                        *
 *      Materials Testing and Evaluation Branch                          *
 * Date: 6 April 1992                                                    *
 *************************************************************************
 * Implementation file for TEK2430A.H                                    *
 *************************************************************************/

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#include "gpib.h"
#include "tek2430a.h"

/*-----------------------------------------------------------------------*/

//Constructor
tek2430A::tek2430A(const gpib &board_in, const int gpib_addr_in)
{
   gpib_addr = gpib_addr_in;
   board = board_in;
}

/*-----------------------------------------------------------------------*/

//Initialize Tektronix 2430A Oscilloscope
unsigned int tek2430A::tek_init()
{
   //Mandatory
   board.device_command("RQS OFF\0", gpib_addr);
   board.device_command("INIT SRQ\0", gpib_addr);

   //Optional -- defaults scope for whole digitized curve, can be changed
   //            with subsequent commands
   board.device_command("PATH OFF\0", gpib_addr);
   board.device_command("START 1\0", gpib_addr);

   char cmd[IO_STR_LEN] = "STOP \0";
   char points[7];
   itoa(MAX_PTS, points, 10);
   strcat(cmd, points);
   unsigned int flag = board.device_command(cmd, gpib_addr);

   return flag;
}

/*-----------------------------------------------------------------------*/

//Set GPIB board
inline void tek2430A::set_board(const gpib &board_in)
{
   board = board_in;
}

/*-----------------------------------------------------------------------*/

//Get GPIB board
inline gpib tek2430A::get_board()
{
   return board;
}

/*-----------------------------------------------------------------------*/

//Set gpib_addr
inline void tek2430A::set_gpib_addr(const int gpib_addr_in)
{
   gpib_addr = gpib_addr_in;
}
```

```
/*-------------------------------------------------------------------------*/

//Get gpib_addr
inline int tek2430A::get_gpib_addr()
{
   return gpib_addr;
}

/*-------------------------------------------------------------------------*/

//Send command to scope
unsigned int tek2430A::tek_command(const char *cmd)
{
   unsigned int flag = board.device_command(cmd, gpib_addr);
   return flag;
}

/*-------------------------------------------------------------------------*/

//Send query to scope, return string data
inline char *tek2430A::s_tek_query(const char *query)
{
   char *data_in = new char[IO_STR_LEN];
   board.device_command(query, gpib_addr);
   board.device_query(IO_STR_LEN, data_in, gpib_addr);
   return(data_in);
}

/*-------------------------------------------------------------------------*/

//Send query to scope, return float data
float tek2430A::f_tek_query(const char *query)
{
   char data_in[IO_STR_LEN];
   board.device_command(query, gpib_addr);
   board.device_query(IO_STR_LEN, data_in, gpib_addr);

   register int c = 0;                    //put pointer to correct place
   while (!isdigit(data_in[c])) c++;
   if ((c != 0) && (data_in[c-1] == '-')) c--;

   return(atof(&(data_in[c])));           //convert to float
}

/*-------------------------------------------------------------------------*/

//Send query to scope, return integer data
int tek2430A::i_tek_query(const char *query)
{
   char data_in[IO_STR_LEN];
   board.device_command(query, gpib_addr);
   board.device_query(IO_STR_LEN, data_in, gpib_addr);

   register int c = 0;                    //put pointer to correct place
   while (!isdigit(data_in[c])) c++;
   if((c != 0) && (data_in[c-1] == '-')) c--;

   return(atoi(&(data_in[c])));           //convert to integer
}

/*-------------------------------------------------------------------------*/

//Capture integer curve from scope
unsigned int tek2430A::capture_curve(const int num_pts, int *curve_out,
   const int i_channel)
{
   if (num_pts > MAX_PTS) return 1;
   char *data_in = new char[MAX_PTS+6];    //scope sends extra bytes
   char cmd[IO_STR_LEN] = "DATA SOURCE:CH\0";
   char channel[3];
   itoa(i_channel, channel, 10);
   strcat(cmd, channel);
```

```
   board.device_command(cmd, gpib_addr);
   board.device_command("PATH OFF\0", gpib_addr);
   board.device_command("START 1\0", gpib_addr);

   strcpy(cmd, "STOP \0");
   char points[7];
   itoa(MAX_PTS, points, 10);
   strcat(cmd, points);
   board.device_command(cmd, gpib_addr);

   board.device_command("DATA ENCDG:RIBINARY\0", gpib_addr);
   board.device_command("CURVE?\0", gpib_addr);
   unsigned int flag = board.device_query((MAX_PTS+6), data_in, gpib_addr);

   for (register int c=0; c<num_pts; c++)        //cast to integer and
      curve_out[c] = data_in[c+3];               //skip extra bytes

   delete data_in;
   return flag;
}

/*-------------------------------------------------------------------------*/

//Get float voltage amplitude from integer amplitude
void tek2430A::convert_to_volts(const int num_pts, float *f_curve,
   const int *i_curve)
{
   float yoff = f_tek_query("WFMPRE? YOFF\0");
   float ymult = f_tek_query("WFMPRE? YMULT\0");

   for (register int c=0; c<num_pts; c++)
      f_curve[c] = (i_curve[c] - yoff) * ymult;  //conversion algorithm
}

/*-------------------------------------------------------------------------*/

//Return sample rate of current signal
float tek2430A::tek_sample_rate()
{
   int num_divisions = 20;
   if (s_tek_query("HORIZONTAL? MODE\0")[0] == 'A')
      return f_tek_query("HORIZONTAL? ASECDIV\0")/(MAX_PTS/num_divisions);
   if (s_tek_query("HORIZONTAL? MODE\0")[0] == 'B')
      return f_tek_query("HORIZONTAL? BSECDIV\0")/(MAX_PTS/num_divisions);
   return 0;
}

/*-------------------------------------------------------------------------*/
```

```
/*****************************************************************************
 *                                DISPLAY.H                                  *
 *****************************************************************************
 * Written by: Bradley M. Taber III                                          *
 * For: U.S. Army Materials Technology Laboratory                            *
 *      Materials Testing and Evaluation Branch                              *
 * Date: 6 April 1992                                                        *
 *****************************************************************************
 * Include file to use screen I/O function on curves                         *
 *****************************************************************************/

#ifndef DISPLAY_H
#define DISPLAY_H

//Screen Output
int display_curve(const int num_pts, const float *dat, const float Y_AMP);

#endif  // DISPLAY_H
```

```
/*****************************************************************************
 *                                DISPLAY.CPP                                *
 *****************************************************************************
 * Written by: Bradley M. Taber III                                          *
 * For: U.S. Army Materials Technology Laboratory                            *
 *      Materials Testing and Evaluation Branch                              *
 * Date: 6 April 1992                                                        *
 *****************************************************************************
 * Implementation file for DISPLAY.H                                         *
 *****************************************************************************/

#include <conio.h>
#include <fstream.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>

#include "display.h"

/*---------------------------------------------------------------------------*/

//Graphically display curve
int display_curve(const int num_pts, const float *dat, const float Y_AMP)
{
   int grph_drvr = DETECT, gmode, errorcode;
   initgraph(&grph_drvr, &gmode, "");
   errorcode = graphresult();
   if (errorcode != grOk)
   {
      cerr << "Graphics Error: " << grapherrormsg(errorcode) << '\n';
      cerr << "Press any key to terminate:";
      getch();
      exit(1);
   }

   setbkcolor(BLUE);
   setcolor(YELLOW);
   cleardevice();
   outtextxy(5, 20, "Display of digitized waveform:");

   for (int c=0; c<num_pts; c++)                  // scale and draw
   {
      float x = (float(c) * (float(640)-float(0))) / float(num_pts);
      float y = 240 - (dat[c] * Y_AMP);
      if (c==0) moveto(x,y);
      else lineto(x,y);
   }

   outtextxy(215, 455, "<HIT ANY KEY TO CONTINUE>");
   getch();
   closegraph();
   return 0;
}

/*---------------------------------------------------------------------------*/
```

```
/*****************************************************************************
 *                              GPIBMAIN.CPP                                 *
 *****************************************************************************
 * Written by: Bradley M. Taber III                                          *
 * For: U.S. Army Materials Technology Laboratory                            *
 *      Materials Testing and Evaluation Branch                              *
 * Date: 6 April 1992                                                        *
 *****************************************************************************
 * Test program for GPIB data acquisition modules                            *
 *****************************************************************************/

#include <conio.h>
#include <dos.h>
#include <fstream.h>

#include "gpib.h"
#include "tek2430a.h"
#include "display.h"

//#define ofil cout                                 //output to screen
ofstream ofil("gpibmain.out");                      //output to file

main()
{
   clrscr();

   //Initialization
   gpib gpib1(DEF_IREQ, DEF_DMA, DEF_MAD, DEF_CIC, DEF_NOB,
      DEF_BA0, DEF_BA1, DEF_ROM_ADDR);
   ofil << "init_gpib() flag = " << gpib1.init_gpib() << '\n';

   tek2430A scope(gpib1, 15);
   ofil << "tek_init() flag = " << scope.tek_init() << '\n';

   cout << "\nSet up scope and hit any key to continue\n\n";
   getch();

/*---------------------------------------------------------------------------*
 *                              Tektronix Workout                            *
 *---------------------------------------------------------------------------*/

   //Commands
   ofil << "bell flag = " << scope.tek_command("BELL\0") << '\n';
   ofil << "message flag 1 = "
        << scope.tek_command("MESSAGE 4: \"THIS IS LINE 4\"\0") << '\n';
   ofil << "message flag 2 = "
        << scope.tek_command("MESSAGE 14: \"THIS IS LINE 14\"\0") << '\n';

   //Queries
   ofil << "double vmin = " << scope.f_tek_query("VMINIMUM?\0") << '\n';
   ofil << "double vmax = " << scope.f_tek_query("VMAXIMUM?\0") << '\n';
   ofil << "double volts/div = " << scope.f_tek_query("CH1? VOLTS\0") << '\n';

   ofil << "int min = " << scope.i_tek_query("MINIMUM?\0") << '\n';
   ofil << "int max = " << scope.i_tek_query("MAXIMUM?\0") << '\n';
   ofil << "int start = " << scope.i_tek_query("START?\0") << '\n';
   ofil << "int stop = " << scope.i_tek_query("STOP?\0") << '\n';

   ofil << "string vminimum = " << scope.s_tek_query("VMINIMUM?\0") << '\n';
   ofil << "string vmaximum = " << scope.s_tek_query("VMAXIMUM?\0") << '\n';
   ofil << "string volts = " << scope.s_tek_query("CH1?\0") << '\n';
   ofil << "string id = " << scope.s_tek_query("ID?\0") << '\n';

   //Data Acquisition
   int *i_curve = new int[MAX_PTS];
   scope.capture_curve(MAX_PTS, i_curve, 1);
   float *f_curve = new float[MAX_PTS];
   scope.convert_to_volts(MAX_PTS, f_curve, i_curve);
   delete i_curve;
```

```
    //Graphical Output to Screen
    cout << "\nHit any key to display graphs\n";
    getch();

    int num_pts = 1024;
    int Y_AMP = 40;
    display_curve(num_pts, f_curve, Y_AMP);
}
```

OUTPUT:

```
init_gpib() flag = 0
tek_init() flag = 0
bell flag = 0
message flag 1 = 0
message flag 2 = 0
double vmin = -1.8
double vmax = 1.88
double volts/div = 1
int min = -45
int max = 49
int start = 1
int stop = 1024
string vminimum = -1.800
string vmaximum = 1.800
string volts = 1,0,0,AC,OFF,OFF
string id = TEK/2430A,V81.1,"21-SEP-87  V1.90 /1.4"
```

BIBLIOGRAPHY

1. 2430A Programmers Reference Guide, Tektronix Publication #070-6338-00. Beaverton, OR: Tektronix Inc., 1988.

2. Atkinson, Lee, Atkinson, Mark. Using Borland C++. Carmel, IN: QUE, 1991.

3. Borland C++ Programmer's Guide. Scotts Valley, CA: Borland International Inc., 1991.

4. Caristi, Anthony, J.. IEEE-488 General Purpose Instrument Bus Manual. San Diego, CA: Academic Press, Inc., 1989.

5. Data Acquisition & Control, KEITHLEY METRABYTE/ASYST/DAC Volume 24. 1991.

6. Dewhurst, Stephen C., Stark Kathy T.. Programming in C++. Englewood Cliffs, NJ: Prentice Hall, 1989.

7. IBM Technical Reference Manual. Boca Raton, FL: IBM Corp., 1984.

8. IE-488 Manual. Taunton, MA: Metrabyte Corporation, 1984.

9. IEEE Standard Digital Interface for Programmable Instrumentation, ANSI/IEEE Std. 488-1978, IEEE. New York, NY 1978.

10. Lippman, Stanley. C++ Primer. Reading, MA: Addison Wesley Publishing Company, 1991.

11. Model, Mitchell L.. Data Structures, Data Abstraction. Draft 2, 1991.

12. Schildt, Herbert. C: The Complete Reference. Berkeley, CA: Osborne McGraw-Hill, 1987.

13. Silberschatz, Abraham, Peterson, James L., Galvin, Peter B.. Operating System Concepts. Reading, MA: Addison-Wesley Publishing Company, 1991.

14. Stover, Allan. ATE: Automatic Test Equipment. New York, NY: McGraw-Hill Company, 1984.

15. Turbo Assembler User's Guide. Scotts Valley, CA: Borland International Inc., 1990.

DISTRIBUTION LIST

No. of Copies	To
1	Office of the Under Secretary of Defense for Research and Engineering, The Pentagon, Washington, DC 20301
	Commander, U.S. Army Laboratory Command, 2800 Powder Mill Road, Adelphi, MD 20783-1145
1	ATTN: AMSLC-IM-TL
1	AMSLC-CT
	Commander, Defense Technical Information Center, Cameron Station, Building 5, 5010 Duke Street, Alexandria, VA 22304-6145
2	ATTN: DTIC-FDAC
1	MIAC/CINDAS, Purdue University, 2595 Yeager Road, West Lafayette, IN 47905
	Commander, Army Research Office, P.O. Box 12211, Research Triangle Park, NC 27709-2211
1	ATTN: Information Processing Office
	Commander, U.S. Army Materiel Command, 5001 Eisenhower Avenue, Alexandria, VA 22333
1	ATTN: AMCSCI
	Commander, U.S. Army Materiel Systems Analysis Activity, Aberdeen Proving Ground, MD 21005
1	ATTN: AMXSY-MP, H. Cohen
	Commander, U.S. Army Missile Command, Redstone Scientific Information Center, Redstone Arsenal, AL 35898-5241
1	ATTN: AMSMI-RD-CS-R/Doc
1	AMSMI-RLM
	Commander, U.S. Army Armament, Munitions and Chemical Command, Dover, NJ 07801
1	ATTN: Technical Library
	Commander, U.S. Army Natick Research, Development and Engineering Center, Natick, MA 01760-5010
1	ATTN: Technical Library
	Commander, U.S. Army Satellite Communications Agency, Fort Monmouth, NJ 07703
1	ATTN: Technical Document Center
	Commander, U.S. Army Tank-Automotive Command, Warren, MI 48397-5000
1	ATTN: AMSTA-ZSK
1	AMSTA-TSL, Technical Library
	Commander, White Sands Missile Range, NM 88002
1	ATTN: STEWS-WS-VT
	President, Airborne, Electronics and Special Warfare Board, Fort Bragg, NC 28307
1	ATTN: Library
	Director, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD 21005
1	ATTN: SLCBR-TSB-S (STINFO)
	Commander, Dugway Proving Ground, Dugway, UT 84022
1	ATTN: Technical Library, Technical Information Division
	Commander, Harry Diamond Laboratories, 2800 Powder Mill Road, Adelphi, MD 20783
1	ATTN: Technical Information Office
	Director, Benet Weapons Laboratory, LCWSL, USA AMCCOM, Watervliet, NY 12189
1	ATTN: AMSMC-LCB-TL
1	AMSMC-LCB-R
1	AMSMC-LCB-RM
1	AMSMC-LCB-RP
	Commander, U.S. Army Foreign Science and Technology Center, 220 7th Street, N.E., Charlottesville, VA 22901-5396
3	ATTN: AIFRTC, Applied Technologies Branch, Gerald Schlesinger
1	Plastics Technical Evaluation Center, (PLASTEC), ARDEC, Bldg. 355N, Picatinny Arsenal, NJ 07806-5000
	Commander, U.S. Army Aeromedical Research Unit, P.O. Box 577, Fort Rucker, AL 36360
1	ATTN: Technical Library

No. of Copies	To
	Commander, U.S. Army Aviation Systems Command, Aviation Research and Technology Activity, Aviation Applied Technology Directorate, Fort Eustis, VA 23604-5577
1	ATTN: SAVDL-E-MOS
	U.S. Army Aviation Training Library, Fort Rucker, AL 36360
1	ATTN: Building 5906-5907
	Commander, U.S. Army Agency for Aviation Safety, Fort Rucker, AL 36362
1	ATTN: Technical Library
	Commander, USACDC Air Defense Agency, Fort Bliss, TX 79916
1	ATTN: Technical Library
1	Clarke Engineer School Library, 3202 Nebraska Ave. North, Ft. Leonard Wood, MO 65473-5000
	Commander, U.S. Army Engineer Waterways Experiment Station, P. O. Box 631, Vicksburg, MS 39180
1	ATTN: Research Center Library
	Commandant, U.S. Army Quartermaster School, Fort Lee, VA 23801
1	ATTN: Quartermaster School Library
	Naval Research Laboratory, Washington, DC 20375
1	ATTN: Code 5830
1	Dr. G. R. Yoder - Code 6384
	Chief of Naval Research, Arlington, VA 22217
1	ATTN: Code 471
1	Edward J. Morrissey, WRDC/MLTE, Wright-Patterson Air Force, Base, OH 45433-6523
	Commander, U.S. Air Force Wright Research & Development Center, Wright-Patterson Air Force Base, OH 45433-6523
1	ATTN: WRDC/MLLP, M. Forney, Jr.
1	WRDC/MLBC, Mr. Stanley Schulman
	NASA - Marshall Space Flight Center, MSFC, AL 35812
1	ATTN: Mr. Paul Schuerer/EH01
	U.S. Department of Commerce, National Institute of Standards and Technology, Gaithersburg, MD 20899
1	ATTN: Stephen M. Hsu, Chief, Ceramics Division, Institute for Materials Science and Engineering
1	Committee on Marine Structures, Marine Board, National Research Council, 2101 Constitution Ave., N.W., Washington, DC 20418
1	Librarian, Materials Sciences Corporation, 930 Harvest Drive, Suite 300, Blue Bell, PA 19422
1	The Charles Stark Draper Laboratory, 68 Albany Street, Cambridge, MA 02139
	Wyman-Gordon Company, Worcester, MA 01601
1	ATTN: Technical Library
	General Dynamics, Convair Aerospace Division, P.O. Box 748, Fort Worth, TX 76101
1	ATTN: Mfg. Engineering Technical Library
1	Department of the Army, Aerostructures Directorate, MS-266, U.S. Army Aviation R&T Activity - AVSCOM, Langley Research Center, Hampton, VA 23665-5225
1	NASA - Langley Research Center, Hampton, VA 23665-5225
1	U.S. Army Propulsion Directorate, NASA Lewis Research Center, 2100 Brookpark Road, Cleveland, OH 44135-3191
1	NASA - Lewis Research Center, 2100 Brookpark Road, Cleveland, OH 44135-3191
	Director, U.S. Army Materials Technology Laboratory, Watertown, MA 02172-0001
2	ATTN: SLCMT-TML
2	Authors

U.S. Army Materials Technology Laboratory,
Watertown, Massachusetts 02172-0001
INSTRUMENT DRIVER FOR GENERAL PURPOSE
INTERFACE BUS (IEEE-488) -
Bradley M. Taber III and
Patrick J. Sincebaugh

Technical Report MTL TR 92-32, May 1992, 49 pp -
illustrations, D/A Project 1L16105AH84

AD____________

UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

General purpose interface bus (GPIB)
Interfaces
Data acquisition

Laboratory automation is invaluable to scientists, engineers, and technicians. It provides a mechanism by which many processes can be interactively controlled. A popular method of connecting and controlling laboratory instrumentation is through the General Purpose Interface Bus (GPIB). The Institute of Electrical and Electronics Engineers (IEEE) standardized this bus under IEEE-488. The objective of this project was to study the IEEE-488 and write instrument drivers in C++ for the GPIB and the Tektronix 2430A Digital Oscilloscope. These drivers contain functions for initialization, command control, and data acquisition. The first section of this paper discusses the IEEE-488 and provides pertinent technical information. The second section of this paper is an instruction manual for the use of the GPIB and Tektronix oscilloscope instrument drivers. Portions of an Input/Output (I/O) module are included to provide a graphical representation for the data acquired with the GPIB and Tektronix modules. A test program is presented which demonstrates the use of these modules. These drivers will be specifically used for ultrasonic data acquisition capabilities for nondestructive testing of materials. The ultrasonic data is generated by an ultrasonic transducer sending a sound pulse through a material. The reflected signal is displayed on a digitizing oscilloscope. The displayed signal must be acquired by the computer for further processing. This system will eventually incorporate robotics for unattended material testing. Human interaction will be kept to a minimum so it will be very important that errors are appropriately handled.

U.S. Army Materials Technology Laboratory,
Watertown, Massachusetts 02172-0001
INSTRUMENT DRIVER FOR GENERAL PURPOSE
INTERFACE BUS (IEEE-488) -
Bradley M. Taber III and
Patrick J. Sincebaugh

Technical Report MTL TR 92-32, May 1992, 49 pp -
illustrations, D/A Project 1L16105AH84

AD____________

UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

General purpose interface bus (GPIB)
Interfaces
Data acquisition

Laboratory automation is invaluable to scientists, engineers, and technicians. It provides a mechanism by which many processes can be interactively controlled. A popular method of connecting and controlling laboratory instrumentation is through the General Purpose Interface Bus (GPIB). The Institute of Electrical and Electronics Engineers (IEEE) standardized this bus under IEEE-488. The objective of this project was to study the IEEE-488 and write instrument drivers in C++ for the GPIB and the Tektronix 2430A Digital Oscilloscope. These drivers contain functions for initialization, command control, and data acquisition. The first section of this paper discusses the IEEE-488 and provides pertinent technical information. The second section of this paper is an instruction manual for the use of the GPIB and Tektronix oscilloscope instrument drivers. Portions of an Input/Output (I/O) module are included to provide a graphical representation for the data acquired with the GPIB and Tektronix modules. A test program is presented which demonstrates the use of these modules. These drivers will be specifically used for ultrasonic data acquisition capabilities for nondestructive testing of materials. The ultrasonic data is generated by an ultrasonic transducer sending a sound pulse through a material. The reflected signal is displayed on a digitizing oscilloscope. The displayed signal must be acquired by the computer for further processing. This system will eventually incorporate robotics for unattended material testing. Human interaction will be kept to a minimum so it will be very important that errors are appropriately handled.

U.S. Army Materials Technology Laboratory,
Watertown, Massachusetts 02172-0001
INSTRUMENT DRIVER FOR GENERAL PURPOSE
INTERFACE BUS (IEEE-488) -
Bradley M. Taber III and
Patrick J. Sincebaugh

Technical Report MTL TR 92-32, May 1992, 49 pp -
illustrations, D/A Project 1L16105AH84

AD____________

UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

General purpose interface bus (GPIB)
Interfaces
Data acquisition

Laboratory automation is invaluable to scientists, engineers, and technicians. It provides a mechanism by which many processes can be interactively controlled. A popular method of connecting and controlling laboratory instrumentation is through the General Purpose Interface Bus (GPIB). The Institute of Electrical and Electronics Engineers (IEEE) standardized this bus under IEEE-488. The objective of this project was to study the IEEE-488 and write instrument drivers in C++ for the GPIB and the Tektronix 2430A Digital Oscilloscope. These drivers contain functions for initialization, command control, and data acquisition. The first section of this paper discusses the IEEE-488 and provides pertinent technical information. The second section of this paper is an instruction manual for the use of the GPIB and Tektronix oscilloscope instrument drivers. Portions of an Input/Output (I/O) module are included to provide a graphical representation for the data acquired with the GPIB and Tektronix modules. A test program is presented which demonstrates the use of these modules. These drivers will be specifically used for ultrasonic data acquisition capabilities for nondestructive testing of materials. The ultrasonic data is generated by an ultrasonic transducer sending a sound pulse through a material. The reflected signal is displayed on a digitizing oscilloscope. The displayed signal must be acquired by the computer for further processing. This system will eventually incorporate robotics for unattended material testing. Human interaction will be kept to a minimum so it will be very important that errors are appropriately handled.

U.S. Army Materials Technology Laboratory,
Watertown, Massachusetts 02172-0001
INSTRUMENT DRIVER FOR GENERAL PURPOSE
INTERFACE BUS (IEEE-488) -
Bradley M. Taber III and
Patrick J. Sincebaugh

Technical Report MTL TR 92-32, May 1992, 49 pp -
illustrations, D/A Project 1L16105AH84

AD____________

UNCLASSIFIED
UNLIMITED DISTRIBUTION

Key Words

General purpose interface bus (GPIB)
Interfaces
Data acquisition

Laboratory automation is invaluable to scientists, engineers, and technicians. It provides a mechanism by which many processes can be interactively controlled. A popular method of connecting and controlling laboratory instrumentation is through the General Purpose Interface Bus (GPIB). The Institute of Electrical and Electronics Engineers (IEEE) standardized this bus under IEEE-488. The objective of this project was to study the IEEE-488 and write instrument drivers in C++ for the GPIB and the Tektronix 2430A Digital Oscilloscope. These drivers contain functions for initialization, command control, and data acquisition. The first section of this paper discusses the IEEE-488 and provides pertinent technical information. The second section of this paper is an instruction manual for the use of the GPIB and Tektronix oscilloscope instrument drivers. Portions of an Input/Output (I/O) module are included to provide a graphical representation for the data acquired with the GPIB and Tektronix modules. A test program is presented which demonstrates the use of these modules. These drivers will be specifically used for ultrasonic data acquisition capabilities for nondestructive testing of materials. The ultrasonic data is generated by an ultrasonic transducer sending a sound pulse through a material. The reflected signal is displayed on a digitizing oscilloscope. The displayed signal must be acquired by the computer for further processing. This system will eventually incorporate robotics for unattended material testing. Human interaction will be kept to a minimum so it will be very important that errors are appropriately handled.